On the Suitability of SHA-3 Finalists for Lightweight Applications

Elif Bilge Kavun, Tolga Yalcin
Chair of Embedded Security
Horst Görtz Institute, Ruhr University - Bochum
Bochum, Germany
{elif.kavun, tolga.yalcin}@rub.de

Abstract. In this study, we investigate the suitability of SHA-3 finalists for lightweight applications. For each finalist, we try to achieve the lowest reported gate count while maintaining a respectable throughput. Our approach differs from all previous SHA-3 implementations, which mainly focus on high performance in terms of throughput. We mainly favor a word-serial approach in our designs to achieve low gate count, where the word size varies from 8 to 64-bits depending on the structure of the hash function and the tradeoff between throughput and area. All hash function cores are realized in Verilog-HDL, synthesized using 90nm UMC CMOS standard cell library and optimized for area for prototyping. A generic FIFO based I/O interface is also built in order to establish data transfer between an external controller and the active hash function core. Results show that, Grøstl has the lowest gate count, while BLAKE gives the best throughput and throughput/area figures. To the best of our knowledge, this is the first comprehensive study on the suitability of SHA-3 finalists for lightweight applications.

Keywords: SHA-3 finalists, BLAKE, Grøstl, JH, Keccak, Skein, serial implementation, lightweight.

1 Introduction

NIST announced a public competition on November 2, 2007 to develop a new cryptographic hash algorithm [1]. The winning algorithm will be named 'SHA-3' and the hash algorithms currently specified in FIPS 180-3, Secure Hash Standard [2], will be augmented. At the moment, the third and final round of the NIST SHA-3 competition is ongoing, in which five finalist algorithms are being considered for the final selection: BLAKE [3], Grøstl [4], JH [5], Keccak [6] and Skein [7]. There have been many studies and discussions on these algorithms since the day they were submitted. Implementation of the algorithms is an important part of these investigations. Several software and hardware implementations deal with effective and high performance realization of the candidates on a wide range of platforms from embedded processors to custom ASICs. However none of them offer a comprehensive study on the suitability of the SHA-3 candidates for lightweight applications.

The term “lightweight” alone covers a very wide range of devices, such as RFID (Radio-Frequency IDentification) tags for identification and tracking purposes using radio waves, smart cards to provide identification, authentication, data storage and application processing, and sensor nodes to gather sensory information. Each of these devices have different requirements in terms of power, operating conditions, speed, area, etc., which means that a study for the lightweight suitability of any security algorithm will have to be done taking into account the specific needs of the application. On the other hand, the most common characteristics of all lightweight applications are the necessity of low cost and sufficiency of low speed. For most lightweight devices, low gate count also corresponds to low power consumption, and speed/throughput is not very important. Therefore, we have decided to limit our focus to low gate count for ASIC implementations.

In today's world, there is a high increase in the utilization of these devices, which results in security and identification problems. The need for lightweight cryptographic hash functions as part of security protocols has been repeatedly expressed. As a result, a few lightweight hash algorithms have recently emerged [8]-[10]. However, these algorithms are quite immature, and their comprehensive analyses are yet to be done.

On the other hand, SHA-3 candidates have already been intensively investigated in term of security, and as a result all but the remaining five finalists have been eliminated. From this point of view, it makes much more sense to study the suitability of these finalists for lightweight applications and, if necessary, come up with suggestions for a possible lightweight extension and/or option in the upcoming SHA-3 standard.

It is the main of target of this study is to present efficient compact implementations of Round 3 SHA-3 candidates offering the lowest possible gate count (and therefore the lowest power consumption), whereas the resultant throughput is still within the limits desirable for lightweight applications. One approach to achieve this target is to replace registers by RAM(s) and implement minimal combinational circuitry necessary for the realization of computational operations. Another approach is to keep the registers, but perform computational operations serially, thereby saving from the combinational logic and interconnection area. We opted for the latter option, mainly because of the non-standard block memory interfaces and performances offered by different process technologies.

We also believe that the structures we propose for each hash function can be easily modified and used within a hybrid approach.

In our study, we chose the 256-bit message digest option for all finalists. Our designs are both suitable for ASIC and FPGA platforms. However, we have used 90 nm UMC CMOS technology for our implementations. Area optimized synthesis results show that Grøstl offers the lowest gate count, while BLAKE offers the best throughput and throughput/area numbers. We have also compared the finalists with each other to observe the overall performance.

The rest of the paper is organized as follows: In Sections 2-6, a brief description of each algorithm is followed by the implementation details of that algorithm, organized alphabetically. Section 7 describes the interface used to connect all hash modules. In section 8, implementation results are provided and the results are compared with previous works. Finally, the paper is concluded with future directions in Section 9.

2 BLAKE

2.1 Algorithm

BLAKE [3] is a family of four hash functions: BLAKE-224, BLAKE-256, BLAKE-384 and BLAKE-512, which follows the HAIFA iteration mode [11]. The compression function depends on a salt and the number of bits hashed so far (as counter): A large inner state is initialized from the initial value, the salt and the counter; and it is injectively updated by message-dependent rounds until it is finally compressed to return the next chain value, as is shown in Figure 1.

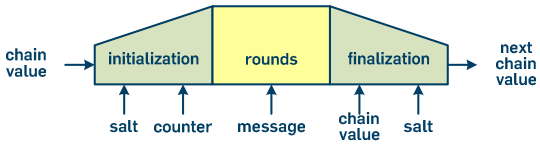

Figure 1. BLAKE compression function

The inner state of the compression function is represented as a 4×4 matrix of words. In one round of BLAKE-256, all four columns and then all four disjoint diagonals are updated independently. In the update of each column or diagonal, two message words are input according to a round-dependent permutation as shown in Figure 2.

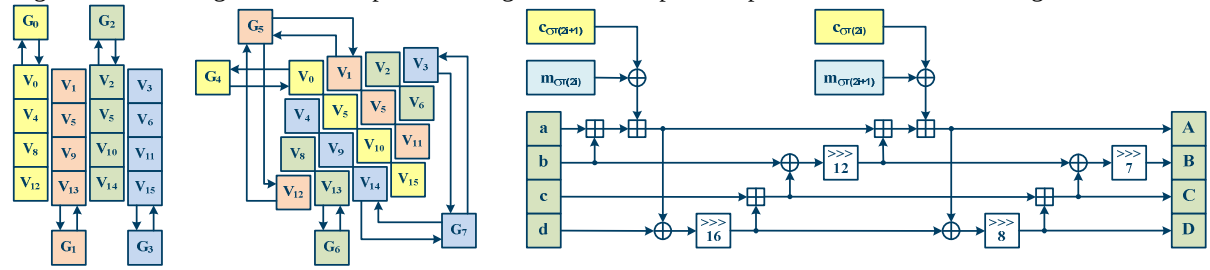

Figure 2. One round of BLAKE and the underlying G_i function

Table 1 shows the specification of BLAKE for 256-bit message digest.

Table 1. BLAKE specifications

Algorithm	Word	Message	Block	Salt	Rounds	Digest
BLAKE-256	32-bit	$< 2^{64}$ - bit	512-bit	128-bit	14	256-bit

2.2 Implementation Details

The serialized architecture for BLAKE is given in Figure 3. The first operation is the initialization, where data is written into the state registers as 32-bit words in 16 cycles. The salt, hash and message registers, which are also shown in Figure 3, store the salt, the hash and the message, respectively. The state words are then processed by the half G_i function block shown in Figure 4, together with the corresponding values from the other registers, and written back on to the state register. The G_i function module operates on each column for $G_{0\text{-}3}$, and then four disjoint diagonals for $G_{4\text{-}7}$ twice because of its 'half' structure. This structure while reducing the area doubles the cycle count.

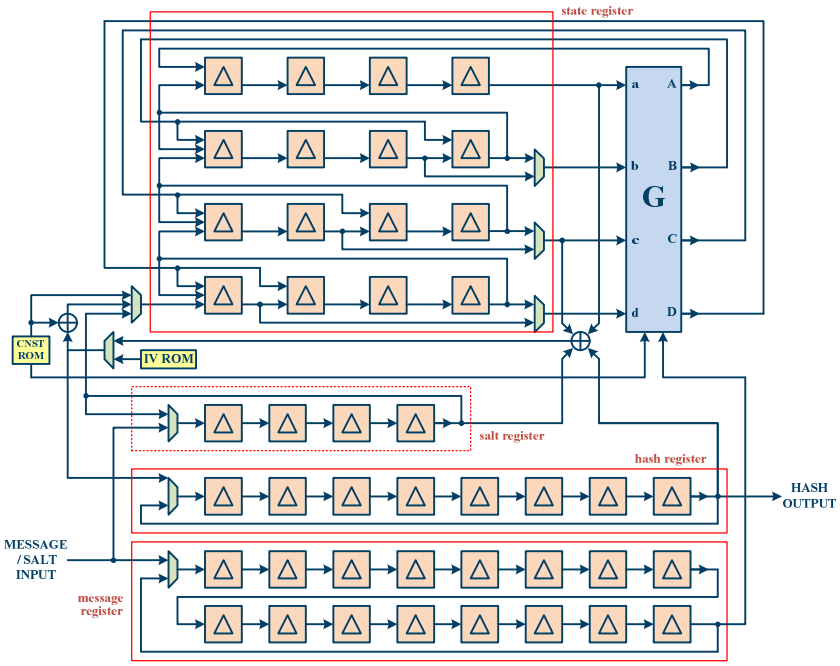

Figure 3. BLAKE serial architecture

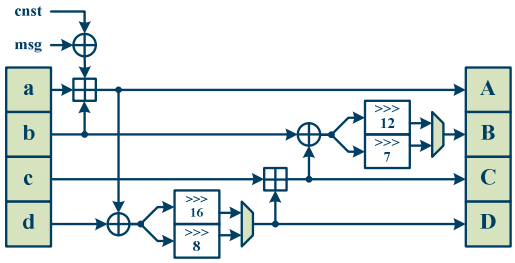

Figure 4. G_i half function

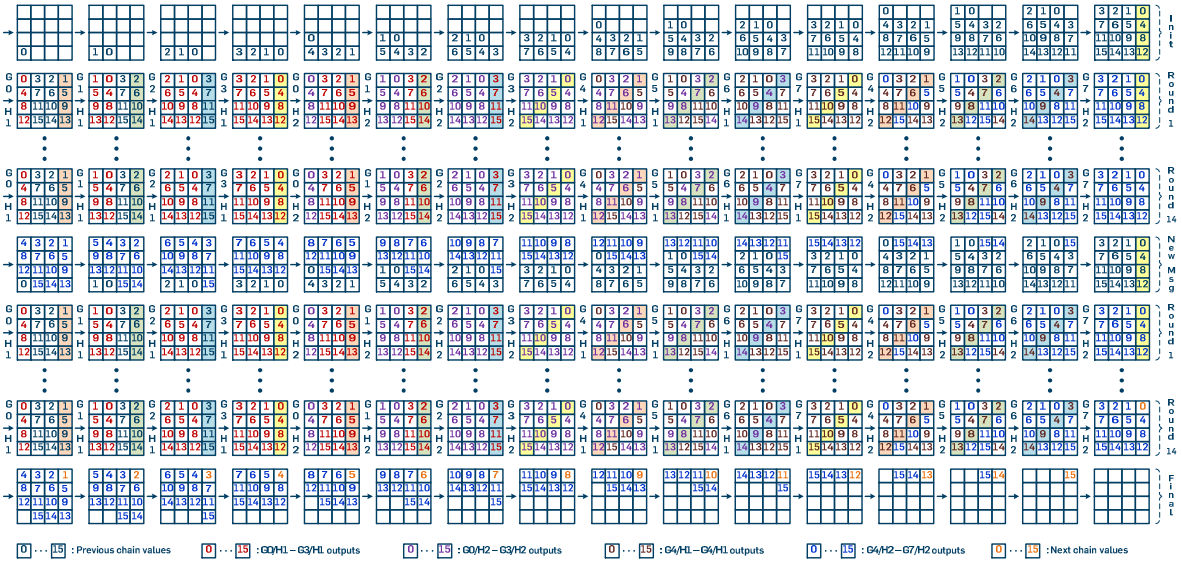

Figure 5. BLAKE serial data flow

As shown in Figure 5, $G_{0\text{-}3}$ is processed at first, in halves (namely H_1 and H_2) followed by the processing of $G_{4\text{-}7}$, again in halves. The multiplexers are switched in order to make sure that the sequence of the serially processed words gives the same result as a parallel implementation. This process is repeated for 14 rounds, and a new message block is injected after the 14th round (if it exists). Injection of message blocks continues until the last block. The finalization process returns the next chain value (or message digest, if it is the last message block).

The whole process is explained in 'phase-round-cycle' concept in Figure 6. In phase-0, the salt is read in 8 cycles. In the following 4 cycles, the length of the message block is read, which is phase-1. Following the length, the first message block is read in phase-2 in 16 cycles. In phase-3, the data processing is performed for 14 rounds (each round in 16 cycles). The next message block is read in phase-4. However, after the last message block, the message digest is written back in the first 8 cycles of phase-4.

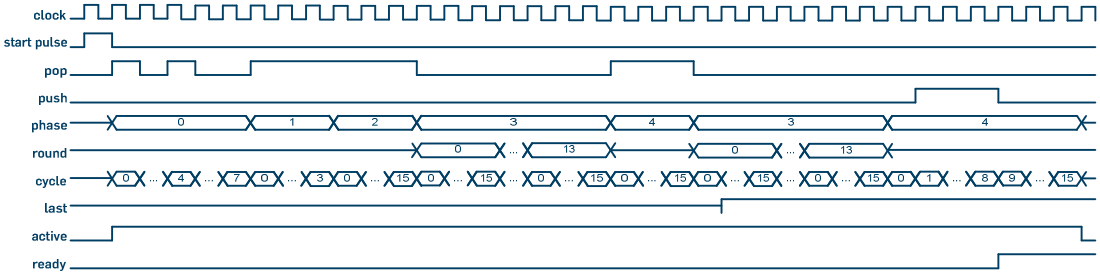

Figure 6. BLAKE timing diagram

3 Grøstl

3.1 Algorithm

Grøstl [4] is a collection of hash functions, which can return message digests from 8 to 512 bits in 8-bit steps. The variant returning n bits is called Grøstl-n. Hashing starts by padding the input message M and splitting it into l-bit message m_1, ... , m_t. Each message block then is processed sequentially by the iterative compression function f, whose other input is the l-bit chaining input with an initial value of $h_0=iv$, as shown in Figure 7. For Grøstl variants with n up to 256 (which covers our case), l is defined to be 512. After the processing of the last message block, the output $H(M)$ of the hash function is computed as $H(M)=\Omega(h_t)$; where Ω is the output transformation, whose output size is n bits, where $n \leq 2l$.

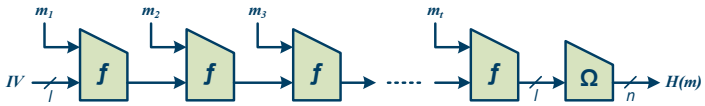

Figure 7. Grøstl compression function

The compression function f is based on two l-bit permutations P and Q, which is defined as $f(h,m) = P(h \oplus m) \oplus Q(m) \oplus h$; and the output function is defined by $\Omega(x)=trunc_n(P(x) \oplus x)$,*where* $trunc_n(x)$ discards all but the trailing n bits of x. Both functions are illustrated in Figure 8. Figure 9 shows details of P and Q permutations.

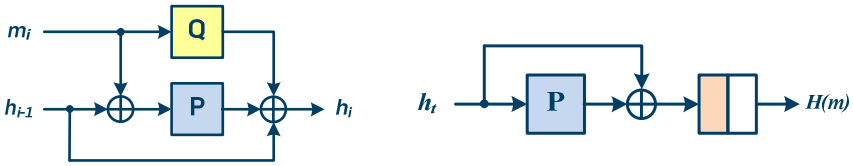

Figure 8. Grøstl construction function *f* (left) and output function *Ω* (right)

Table 2 shows the specification of Grøstl for 256-bit message digest.

Table 2. Grøstl specifications

Algorithm	Word	Message	Block	Salt	Rounds	Digest
Grøstl-256	32-bit	$< (2^{73} - 577)$ - bit	512-bit	10	256-bit	Grøstl-256

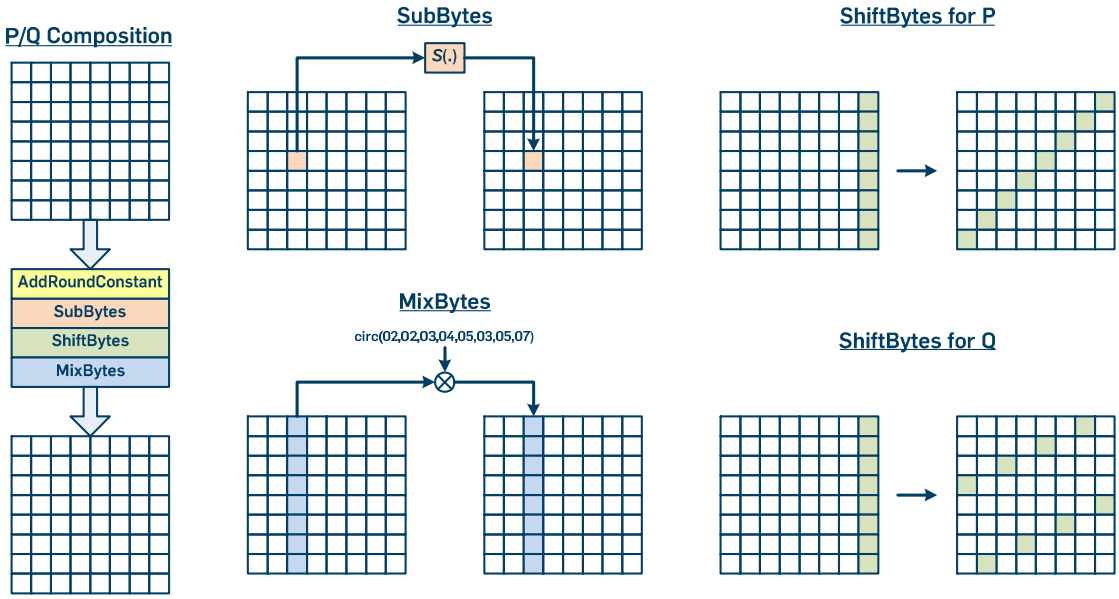

Figure 9. *P* and *Q* permutations

3.2 Implementation Details

The serialized architecture for Grøstl is shown in Figure 10. There exists only a single block for both *P* and *Q* operations in order to save area, which also allows us to use the same block for both *f* and *Ω* functions. For the *f* function, message and previous hash result (which is *iv* at the first round) are selected as input. For the output function omega, the only input comes from hash register and zero is selected instead of the message.

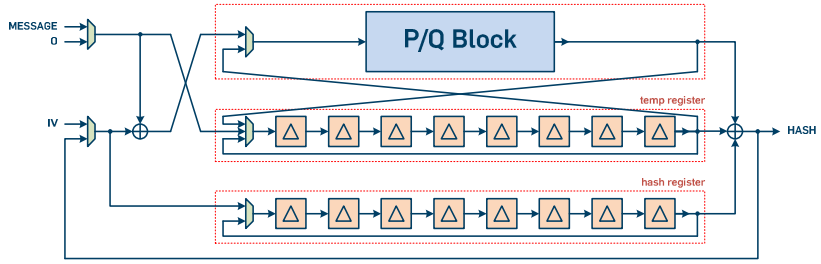

Figure 10. Grøstl serial architecture

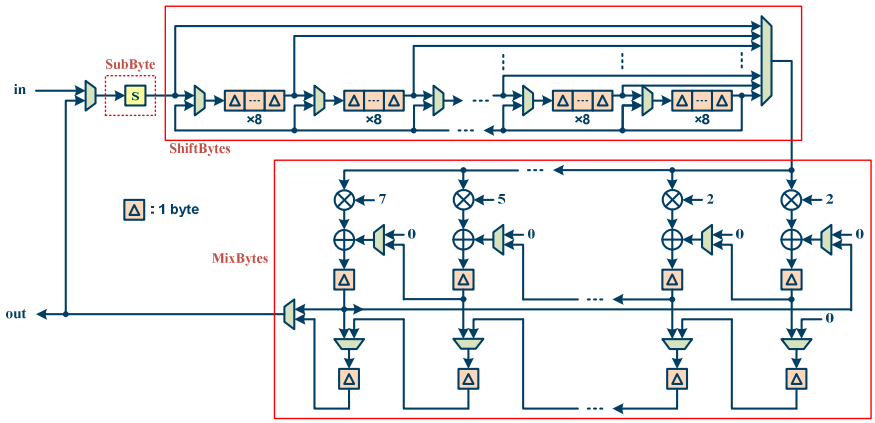

Figure 11. Details of *P/Q* block

While the message is processed inside the *P/Q* module in *P* mode, it is also stored inside the temp register. In the *Q* mode, the result of *P* is stored inside the temp register while the message is restored. It is then processed in *Q* mode, and its result is combined with the *P* result (restored from the temp register) and the previous hash value. The detailed block diagram of *P/Q* module is shown in Figure 11. It basically implements a modified version of the serial AES-like data flow in [12] via SubBytes, ShiftBytes and MixBytes functions. The data flow for a 4x4 toy version of ShiftBytes and MixBytes are given in Figure 12, note that ShiftBytes operation is different for *P* and *Q*.

The whole process is explained in 'phase-half round-round-cycle' concept in Figure 13. In phase-0, the length is read in 10 cycles. Phase-1 is for reading the initialization vector *iv*. Following this, the message blocks are read and processed. Finally, in phase-3, the message digest is written back during phase-3.

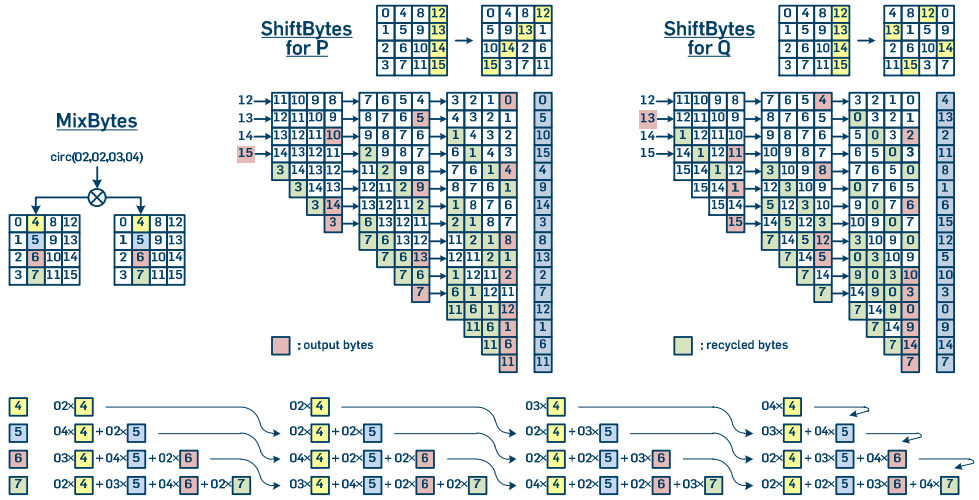

Figure 12. Data flow for 4x4 toy version

Figure 13. Grøstl timing diagram

4 JH

4.1 Algorithm

JH [5] is a family of four hash algorithms – JH-224, JH-256, JH-384 and JH-512. In the design of JH, a compression function is constructed from a large block cipher with constant key. Generalized *d*-dimensional AES design methodology is applied in the design of the large cipher. In our case of 256-bit digest, *d* is set to 8, hence the compression function is named as F_8. It sequentially processes the padded and split message blocks m_1, ... , m_t, starting with an initial vector (*iv*), as shown in Figure 14.

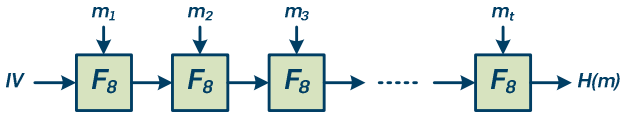

Figure 14. JH compression function

F_8 is bijective due to the block cipher, whose block size is *2m* bits. Its structure is shown in Figure 15 together with the internal function E_8. The *2m*-bit hash value $H_{(i-1)}$ and the *m*-bit message block $M_{(i)}$ are compressed into the *2m*-bit *H(i)*. E_8 is also bijective and applies SPN and MDS to the bit array. MDS is applied before the first and after the last rounds. The round function R_8 consists of an S-box layer (selected via round constants), a linear transformation layer (applied on bytes) and a permutation layer P_8 (composed of three permutations), whose details can be seen in Figure 16. R_8 is repeated 42 times.

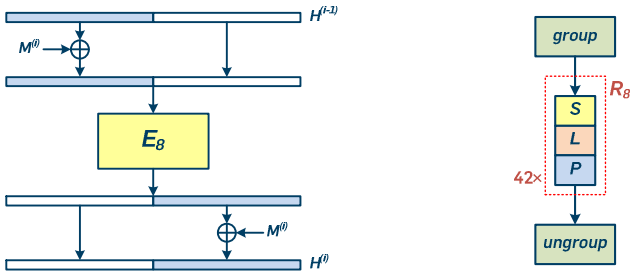

Figure 15. Structure of F_8 compression function (left) and E_8 function (right)

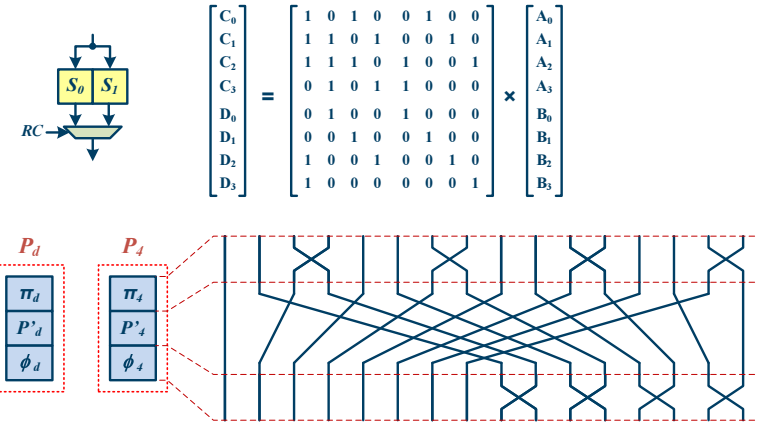

Figure 16. Three layers of round function

Table 3 shows the specification of JH for 256-bit message digest.

Table 3. JH specifications

Algorithm	Word	Message	Block	Salt	Rounds	Digest
JH-256	32-bit	$< 2^{64}$ - bit	512-bit	42	256-bit	JH-256

4.2 Implementation Details

The serialized architecture for JH is given in Figure 17. 32-bit datapath is used in the serialized implementation of JH. The state register is filled with the sum (XOR) of the initialization vector and the message block at the beginning of the process, while the message is also backed up in the message register for post-processing. Upon completion of the rounds, the output of the E_8 block is combined with the backed up message to form the next value of the state register (hash), which in turn is summed with the next message block. This process continues until all the message blocks are processed.

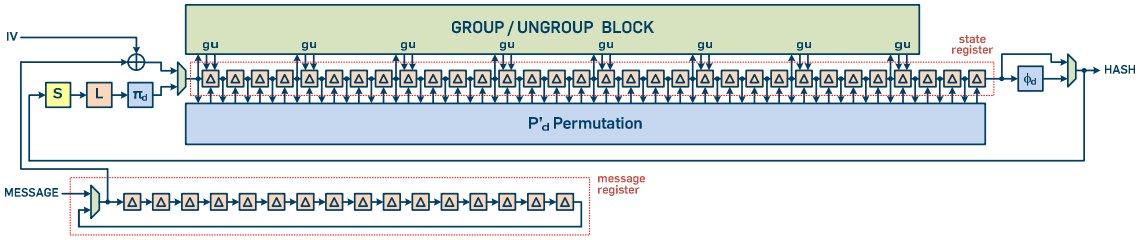

Figure 17. JH serial architecture

The group/de-group block realizes the grouping and de-grouping steps of E_8 function. It only performs grouping/de-grouping at word level. Instead of implementing bit-level grouping/de-grouping, E_8 round function is modified in order to support operation on the word level grouped input and produce output compatible with word level de-grouping. Serialized E_8 round function consists of an S-box, the linear transformation block, and the π_d, P_d' and ϕ_d partial permutation blocks. All, except the P_d'-module, operate on 32-bits.

The serial data flow of JH is shown in Figure 18. It starts with the grouping round, which lasts for 32 cycles. This round is followed by R_8 round function for 42 rounds (each of them is again 32 cycles). After R_8 process, de-grouping round is performed. These grouping and de-grouping operations result in two additional rounds, which make 44 rounds in total. For the last message block, one extra quarter round is required for squeezing the output.

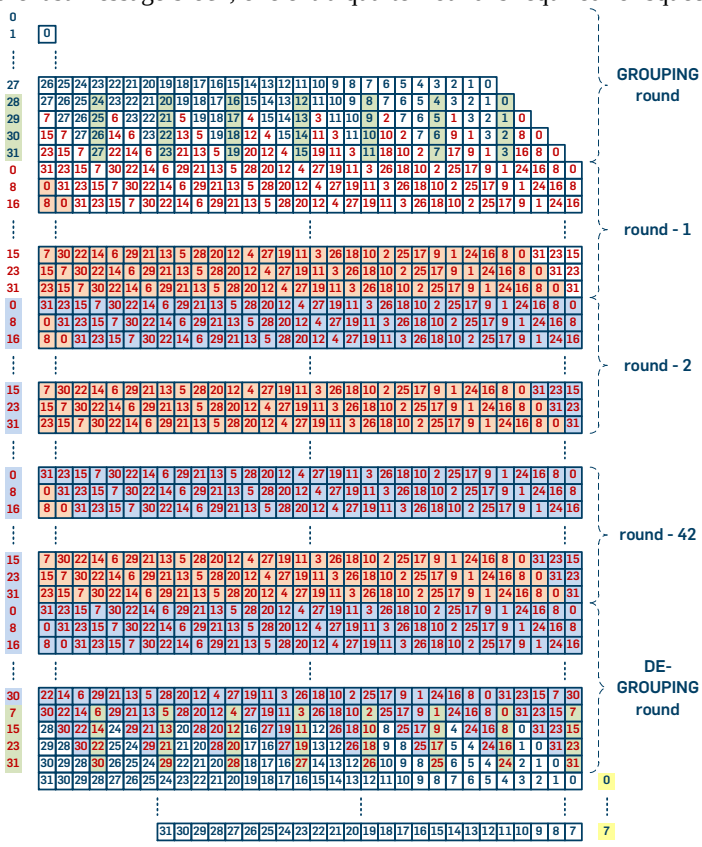

Figure 18. JH serial flow

The whole process is explained in ‘phase-round-cycle’ concept. In phase-0, the length of the message block is read. Then, in phase-1, initialization vector is read and stored in state register. In phase-2, the message blocks are read in every round-0 and these message blocks are processed from round-1 to round-44. Also, the message digest is written back in round-44 of the last message block, again in phase-2. This scheme can be seen in Figure 19.

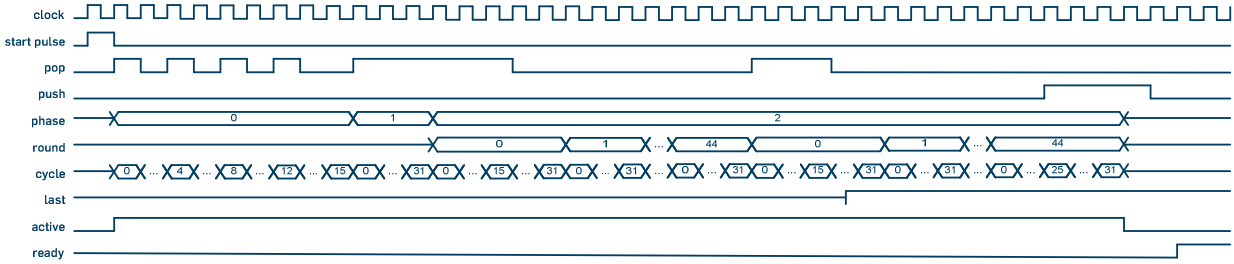

Figure 19. JH timing diagram

5 Keccak

5.1 Algorithm

Keccak [6] is a family of hash functions based on the sponge construction [13]. The fundamental function is the Keccak-*f[b]* permutation, which consists of a number of simple rounds with logical operations and bit permutation. $b \in \{25,50,100,200,400,800,1600\}$ is both width of the permutation, and width of the state in the sponge construction. In our work, we concentrate on Keccak-*f[1600]* with 256-bit message digest.

The state of Keccak is organized in 5×5 lanes, each with w-bits, where $w \in \{1,2,4,8,16,32,64\}$, and $b=25w$. The Keccak*[r,c,d]* sponge function (Figure 20) is obtained by applying the sponge construction to Keccak-*f[r+c]* with the parameters capacity c, bit rate r (which are 512 and 1088, respectively, for Keccak-*f[1600]*). The flow of Keccak-*f* and the details of the steps are given in Figure 21. The number of rounds n_r depends on the permutation width which is calculated by $n_r = 12+2\times l$, where $2^l = w$. This yields 24 rounds for Keccak-*f[1600]*.

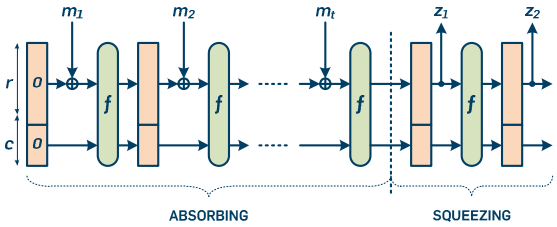

Figure 20. Sponge construction of Keccak

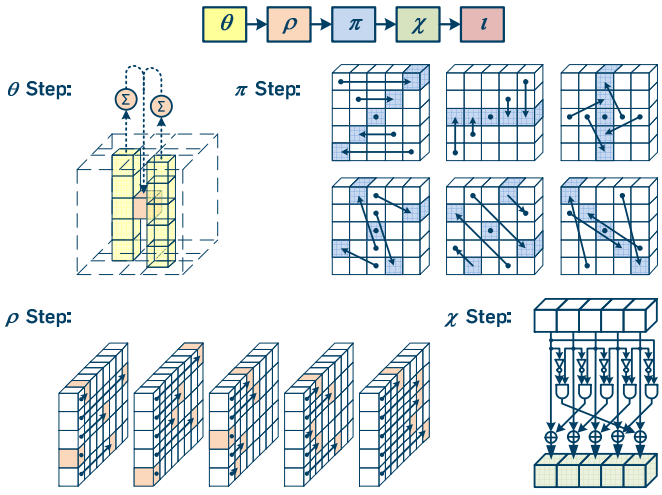

Figure 21. Keccak-*f* function and steps of the function

Table 4 shows the specification of Keccak-*f[1600]* for 256-bit message digest.

Table 4. Keccak specifications

Algorithm	Word	Message	Block	Salt	Rounds	Digest
Keccak-256	64-bit	$< 2^{128} - \text{bit}$	1088-bit	24	256-bit	Keccak-256

5.2 Implementation Details

The serialized architecture for Keccak is given in Figure 22. In the serial design, data is processed in lanes, which is 1/25 of the whole state. The state registers, numbered 24-0, are used to store the internal state, and the four summation registers (rightmost registers numbered 4-0) store the row sums. The operational blocks which implement a Keccak round are the θ, ρ, π, χ, ι-modules. All, but π-module, operate on a single lane. π-step is executed in parallel on all 25 lanes. It is a fixed permutation operation, and the only area cost comes from additional multiplexers and routing. There is additional area cost caused by sum registers (required for θ-step) and two temporary registers (required for χ-step). However, this additional area is compensated by the huge area saving of the serialized processing and the resulting single lane combinational blocks.

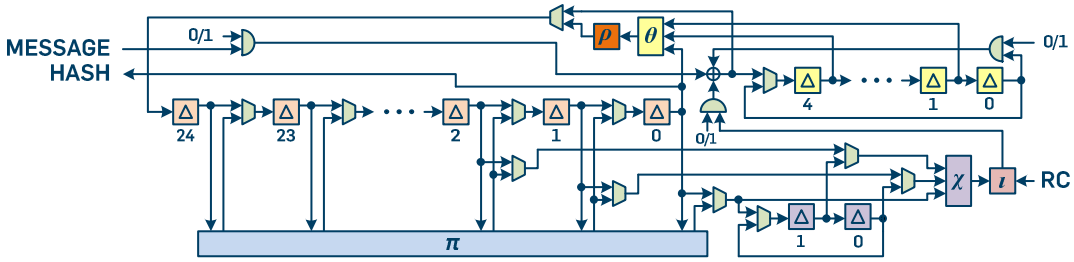

Figure 22. Keccak serial architecture

The processing starts with round-31, where the length of the message block is read. Then round-0 comes, where data is written in lanes into the state registers and each row sum is accumulated inside the sum registers. The first incoming data is lane(0,0) and shifted into state register 24 while sum register 4 is filled with the same value. In the next cycle, state register 24 is shifted into state register 23 and filled with the incoming lane(1,0). In parallel, sum register 4 is shifted into sum register 3, and re-initialized with lane(1,0). At the end of the first 5 cycles, the first 5 lanes of data are in state registers 24 to 20, while sum registers 4 to 0 have the first lanes of each column. In the following cycles, incoming data are added on to sum registers and shifted into the state registers. At the end of the first 25 cycles, state registers contain the full state and sum registers contain the row sums.

Starting with the next cycle, θ and ρ operations are run in parallel from lane(0,0) until lane(4,4), covering the whole state. These operations are completed in 25 cycles. It is followed by another 25 cycles, where π, χ and ι operations are performed. Since π can only be executed on the whole state, it is done in parallel with the first lane of χ. ι operation (round constant addition) is also done in the same cycle. In the following 24 cycles, χ operation is performed on the remaining lanes, completing the first round. Each of these 25 cycles are named as 'half rounds'. The row summations for the following round are also performed in parallel with π, χ and ι operations of the current round, as an additional optimization. A full round takes 50 cycles to complete.

At the end of the 24 rounds, the second half round of the 'last' round is used for 'squeezing' the message digest. The timing diagram in Figure 23 shows the round, half round and cycles for processing of two message blocks.

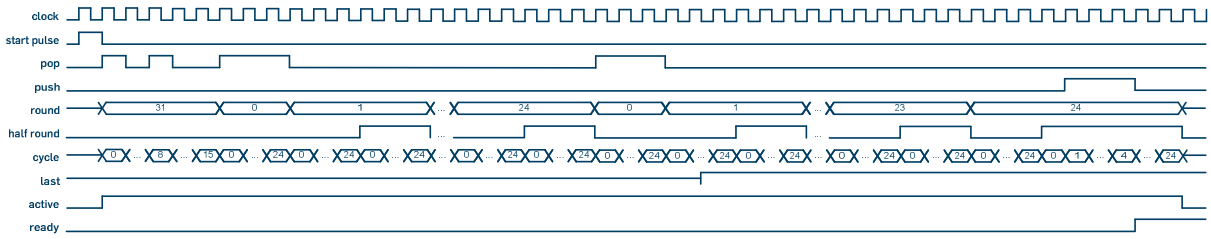

Figure 23. Keccak timing diagram

The whole data processing in each half round is explained by a 3x3 lanes toy-version of Keccak in Figure 24, instead of the actual 5x5 lanes configuration.

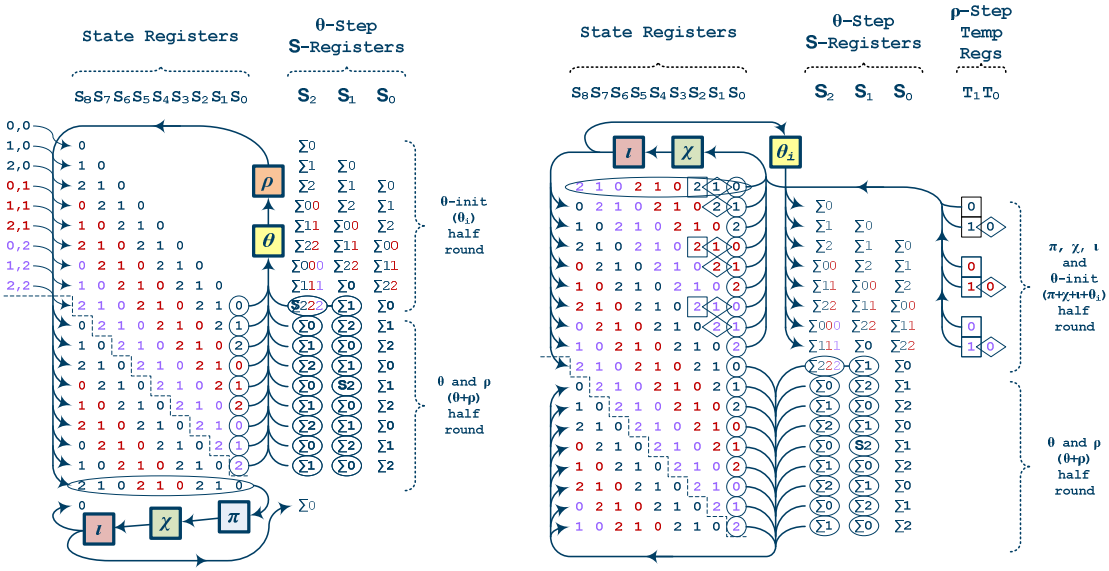

Figure 24. Keccak data flow

6 Skein

6.1 Algorithm

Skein [7] is a family of hash functions with three different internal state sizes: 256, 512 and 1024 bits, where Skein-512 is the primary hash function and can be used for all current hashing applications. Skein hash function is build out of a tweakable block cipher (ThreeFish), which allows hashing configuration data along with the input text in every block, making every instance of the compression function unique. In addition to ThreeFish tweakable block cipher (256, 512 and 1024-bit block sizes) at the core, Skein is built is built up of a unique block iteration (UBI), which maps an arbitrary input size to a fixed output size, and an optional argument system to allow supporting different optional features. The normal (straightforward) hashing option we use can be seen in Figure 25. First block is for configuration, following instances are for message processing, and the last block is for output processing.

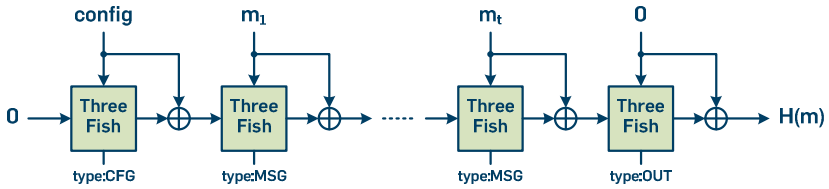

Figure 25. Skein normal hashing scheme

ThreeFish tweakable block cipher is defined for 256, 512 and 1024-bit block sizes. The key is the same size as the block, and the tweak value is 128 bits for all block sizes. Each one of Skein-512's 72 rounds consists of four MIX functions followed by a permutation of the eight 64-bit words. A subkey is added every four rounds. The word permutation is the same for every round, and the rotation constants repeat every eight rounds. A key schedule is also performed for generating subkeys from the original key and the tweak. Figure 26 shows ThreeFish-512 construction for four rounds together with the internal details of the MIX function, which is an add-rotate-XOR (ARX) construction.

Table 5 shows the specification of Skein for 256-bit message digest.

Table 5. Skein specifications

Algorithm	Word	Message	Block	Salt	Rounds	Digest
Skein-256	32-bit	$< 2^{64}$ - bit	512-bit	72	256-bit	Skein-256

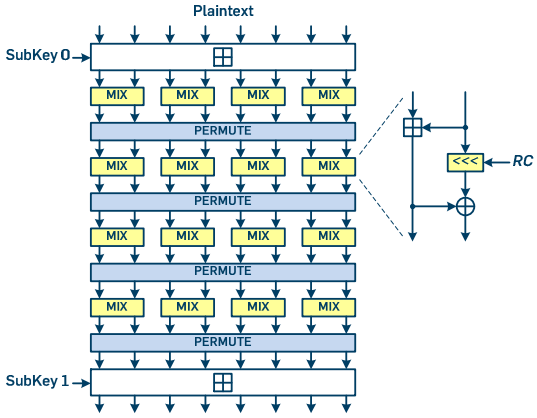

Figure 26. Four rounds of ThreeFish-512

6.2 Implementation Details

The serialized architecture for Skein is given in Figure 27. In round-0, the rightmost eight key expansion registers are filled with input key in 8 cycles, while all input key words are accumulated in the leftmost key register. This practically implements the key expansion process defined for ThreeFish. Following this round, state register is filled the sum of the input message block and the subkey generated in the previous round. In parallel, key expansion process continues within the key registers. At the same time, message block is backed up inside the message register for post-processing following the completion of all ThreeFish rounds.

ThreeFish processing inside the state register is done via a 128-bit MIX block and a fully parallel 512-bit permutation block, which is a fixed 64-bit word based permutation. Its only cost is multiplexers. The 128-bit MIX block requires an additional 64-bit temporary register in order to collect 128-bits of data. At the end of round-42, ThreeFish operation is completed, and round-43 is used to add the stored messages on to the ThreeFish result (UBI operation) in order to obtain the next state of the hash. The operation is repeated until all message blocks are processed. The serial data flow of Skein is shown in Figure 28.

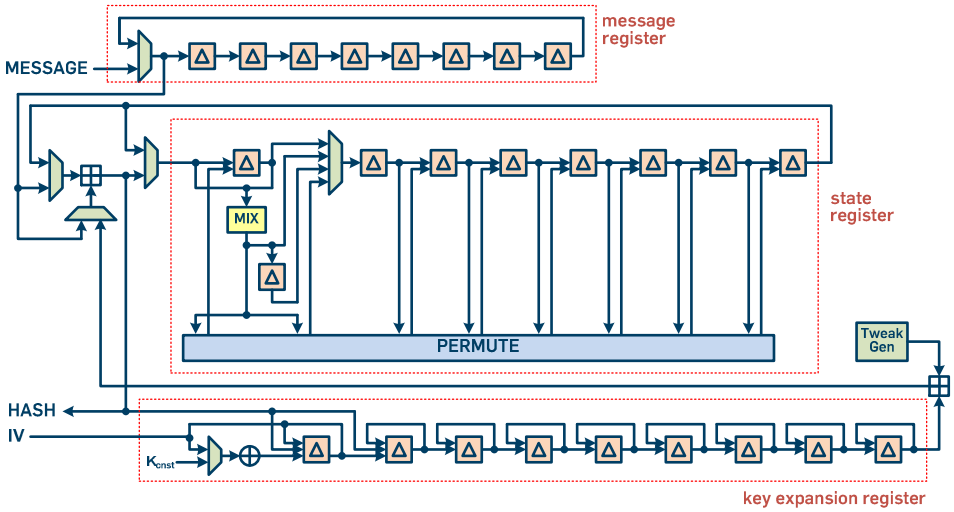

Figure 27. Skein serial architecture

The whole process is explained in ‘phase-round-cycle’ concept. In phase-0, the length of the message block is read. Then, in phase-1, 512-bit initialization vector is directly read from RAM, which makes additional ThreeFish run not necessary. In phase-2, the message blocks are read and processed. Following this, hash value is updated in phase-3. Phase-2 and phase-3 are repeated in series, until all message blocks are processed. After the processing of the last message block, the message digest is written back in that block's phase-3. This scheme can be seen in Figure 29.

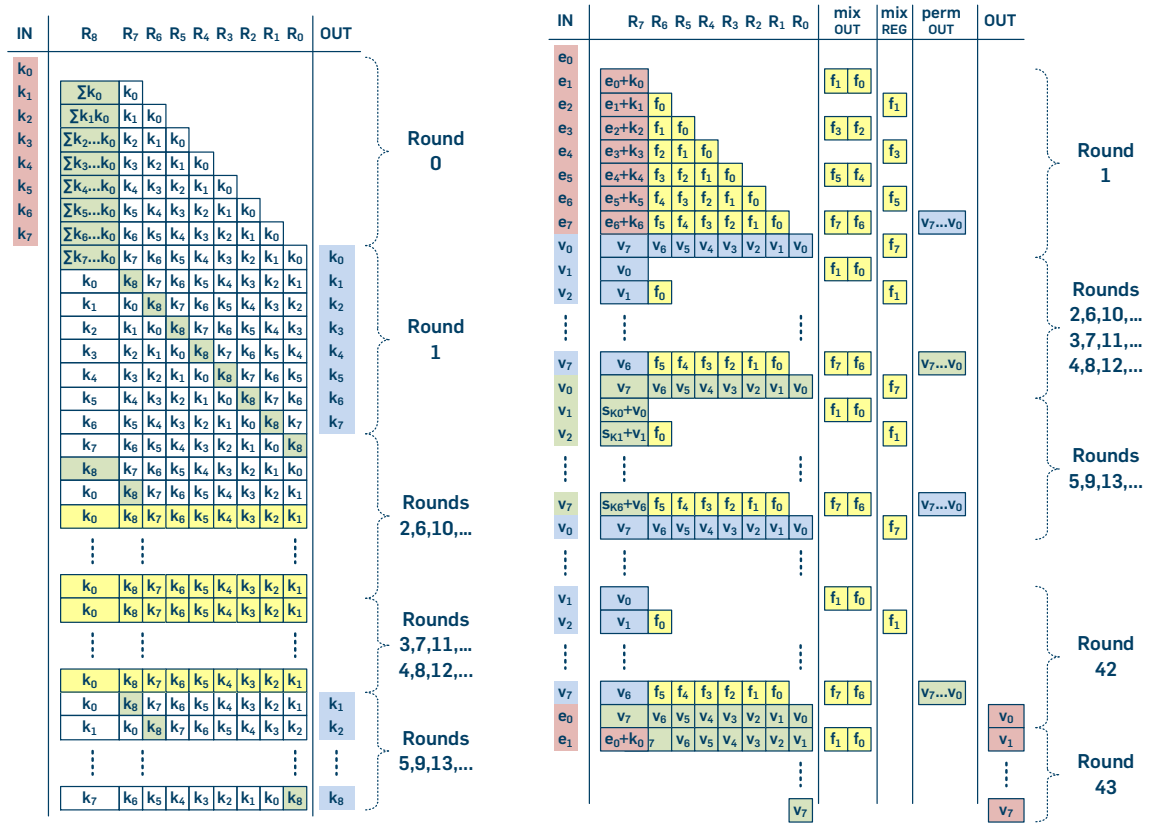

Figure 28. Skein serial flow

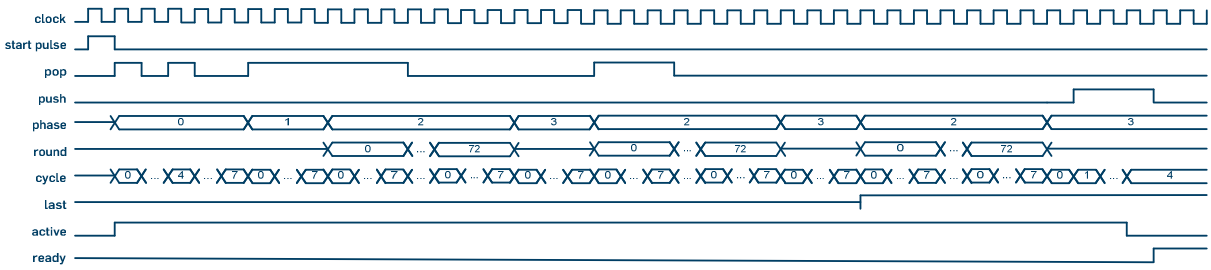

Figure 29. Skein timing diagram

7 Interface

All five hash modules are connected to a 32-bit FIFO based I/O interface module for connection to the external world in the future prototype IC. Internal interface with modules is 64-bit for Keccak and 32-bit for all other blocks. The FIFO is organized as an even/odd couple in order to provide 64-bits necessary for the Keccak block (both FIFOs active) and 32-bits for the others (odd FIFO active in odd cycles, even FIFO active in even cycles). A simple REQuest/ACKnowledge signaling scheme is implemented, where REQ signal is set when FIFO is almost empty and ACK is set when the result is ready. 2016 bytes of memory exist for MESSAGE/DATA and 32 bytes are present for HASH result (message digest). The architecture enables only the selected module, and disables the others via clock gating. This interface architecture is shown in Figure 30.

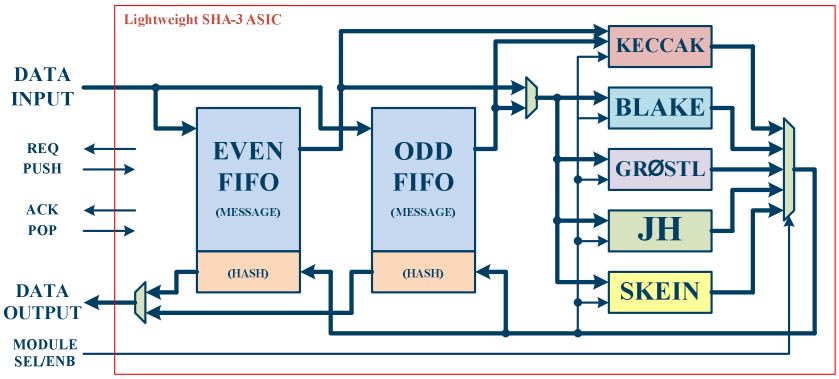

Figure 30. Interface model

8 Results and Discussion

In our study, we achieved better results than most of the previous works in terms of area and throughput. Grøstl and BLAKE give the best gate counts. Best throughput numbers are presented by BLAKE and Keccak, while the best results are provided by BLAKE and Keccak in terms of throughput/area.

Note that, except for Keccak, all hash functions have half the internal state size with respect to 512-bit message digest option. Such a normalization for Keccak will result in Keccak-800-256, and will yield the best gate count and worst throughput. It is also worth mentioning that the throughput of Grøstl can be quadrupled at the expense of an additional 2KGE (estimated), making it the second best in terms of throughput, while preserving its top position with the smallest area.

Table 6 lists our results for all finalists as well as comparison with previous works.

Table 6. Comparison of our work with previous works

Reference	Tech	Area (KGE)	Message Block Size (bits)	Frequency (MHz)	Cycles per Block	Tput (Kbps @ 100KHz)	Tput / Area (bps per GE)
BLAKE [14]	180nm	13.58	512	215	816	63	4.64
BLAKE [14]	180nm	8.6[a]	512	100	N.A.	63	7.33
Our BLAKE	**90nm**	**11.3**	**512**	**N.A.**	**240**	**213**	**18.88**
Grøstl [15]	350nm	14.622	512	56	N.A.	261	17.85
Our Grøstl	**90nm**	**9.2**	**512**	**N.A.**	**1280**	**40**	**4.32**
JH [16]	180nm	58.832	512	380.22	39	1313	22.32
JH [17]	90nm	31.864	512	353	N.A.	1314	41.24
Our JH	**90nm**	**13.6**	**512**	**N.A.**	**1440**	**36**	**2.61**
Keccak [6]	130nm	9.3[b]	1088	200	5160	20	2.15
Our Keccak	**90nm**	**15.2**	**1088**	**N.A.**	**1200**	**91**	**5.96**
Skein [15]	350nm	12.890[c]	512	80	N.A.	25	1.94
Skein [17]	90nm	22.562[d]	512	50	10	2694	119.40
Our Skein	**90nm**	**15.5**	**512**	**N.A.**	**592**	**86**	**5.58**

a) This compact core uses an external memory to hold the message block and does not provide salted hashing.
b) This value includes the area of the RAM. With external RAM, the coprocessor uses 5kGE (as reported in the Keccak main document). Including the area of the RAM yields 9.3kGE.
c) Skein-256-256.
d) Skein-512-256.

9 Conclusion and Future Work

'Lightweight' is the rising star of cryptography. However, since existing security standards and primitives are most of the time not suitable for deployment in lightweight devices, there have already been several creative implementations of these standards targeted for lightweight applications. Furthermore an increasing number of new lightweight algorithms have been proposed. While these algorithms have mostly focused on block ciphers, researchers have recently focused on lightweight hash functions as well. Unfortunately, these studies have so far taken a path completely independent of the ongoing SHA-3 standardization process, where the suitability for lightweight applications issue is neglected. We believe that the two efforts should somehow be combined or at least associated.

Such an association can only be possible after a thorough suitability analysis of the SHA-3 finalists for lightweight applications. The term 'lightweight' alone covers a very wide range, such as lightweight in terms of area, speed, power consumption, energy consumption, or a combination of these, depending on the specific application. Therefore, we have limited our focus on the lightweight for area, which also results in lightweight for average power consumption in most applications; and tried to reach the lowest possible recorded gate counts for all five finalists. Use of block memories is avoided for compatibility on different platforms. We have been successful in reaching our target of lowest gate count, and even managed to surpass some of the recently proposed lightweight hash functions in terms of compactness and throughput.

The next step in our study is the prototyping of the lightweight versions of the finalists. This will also allow us to perform a comprehensive power analysis. For the target prototype, we have already implemented a generic FIFO based interface in order to allow data transfer between an external controller and the SHA-3 finalists. We also plan to implement our lightweight circuits on different FPGA platforms, and analyze their side-channel attack resistance first on the FPGA implementations, then on the prototyped ICs.

10 References

[1] NIST, National Institute of Standards and Technology [docket no.: 070911510-7512-01] Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family, Federal Register, November 2007.

[2] NIST, Federal Information Processing Standards Publication 180-3, Secure Hash Standards (SHS), October 2008.

[3] J.-P. Aumasson, L. Henzen, W. Meier, R. C.-W. Phan, SHA-3 Proposal BLAKE, submission to NIST (Round 3), 2010.

[4] P. Gauravaram, et al., Grøstl – A SHA-3 Candidate, January 2011.

[5] H. Wu, The Hash Function JH, submission to NIST (Round 3), 2011.

[6] G. Bertoni, J. Daemen, M. Peeters and G. Van Assche, The Keccak Sponge Function Family, May 2011.

[7] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, J. Walker, The Skein Hash Function Family, submission to NIST (Round 3), 2010.

[8] H. Yoshida, D. Watanabe, K. Okeya, J. Kitahara, J. Wu, O. Kucuk and B. Preneel, MAME: A Compression Function with Reduced Hardware Requirements, CHES 2007, volume 4727 of LNCS, pages 148-165, Springer, 2007.

[9] J.-P. Aumasson, L. Henzen, W. Meier and M. Naya-Plasencia, Quark: A Lightweight Hash, CHES 2010, volume 6225 of LNCS, pages 1-15, Springer, 2010.

[10] J. Guo, T. Peyrin and A. Poschmann, The PHOTON Family of Lightweight Hash Functions, CRYPTO 2011, volume 6841 of LNCS, pages 222-239, Springer, 2011.

[11] E. Biham, O. Dunkelman, A Framework for Iterative Hash Functions - HAIFA, ePrint report 2007/278, 2007.

[12] Hämäläinen, P., Alho, T., Hännikäinen, M., Hämäläinen, T.D., Design and Implementation of Low-Area and Low-Power AES Encryption Hardware Core, Nineth Euromicro Conference on Digital System Design, IEEE Computer Society, Dubrovnik, 2006.

[13] G. Bertoni, J. Daemen, M. Peeters and G. Van Assche, Sponge Functions, Ecrypt Hash Workshop 2007.

[14] L. Henzen, J.-P. Aumasson, W. Meier, R. C.-W. Phan, VLSI Characterization of the Cryptographic Hash Function BLAKE, IEEE Transactions on VLSI Systems, Volume: 19, Issue: 10, Pages: 1746-1754, Oct. 2011.

[15] S. Tillich, M. Feldhofer, W. Issovits, T. Kern, H. Kureck, M. Mühlberghuber, G. Neubauer, A. Reiter, A. Köfler, and M. Mayrhofer, Compact Hardware Implementations of the SHA-3 Candidates ARIRANG, BLAKE, Grøstl, and Skein, Cryptology ePrint Archive: Report 2009/349, 2009.

[16] S. Tillich, M. Feldhofer, M. Kirschbaum, T. Plos, J.-M. Schmidt, and A. Szekely, High-speed Hardware Implementations of BLAKE, Blue Midnight Wish, CubeHash, ECHO, Fugue, Grøstl, Hamsi, JH, Keccak, Luffa, Shabal, SHAvite-3, SIMD, and Skein, Cryptology ePrint Archive, Report 2009/510, 2009.

[17] M. Knezevic, K. Kobayashi, J. Ikegami, S. Matsuo, A. Satoh, U. Kocabas, J. Fan, T. Katashita, T. Sugawara, K. Sakiyama, I. Verbauwhede, K. Ohta, N. Homma, T. Aoki, Fair and Consistent Hardware Evaluation of Fourteen Round Two SHA-3 Candidates, IEEE Transactions on VLSI Systems, PP(99), pp. 1-13, 2011.